Recursion and Backtracking

Prof. Dr. Murat GÖK

murat.gok@yalova.edu.tr

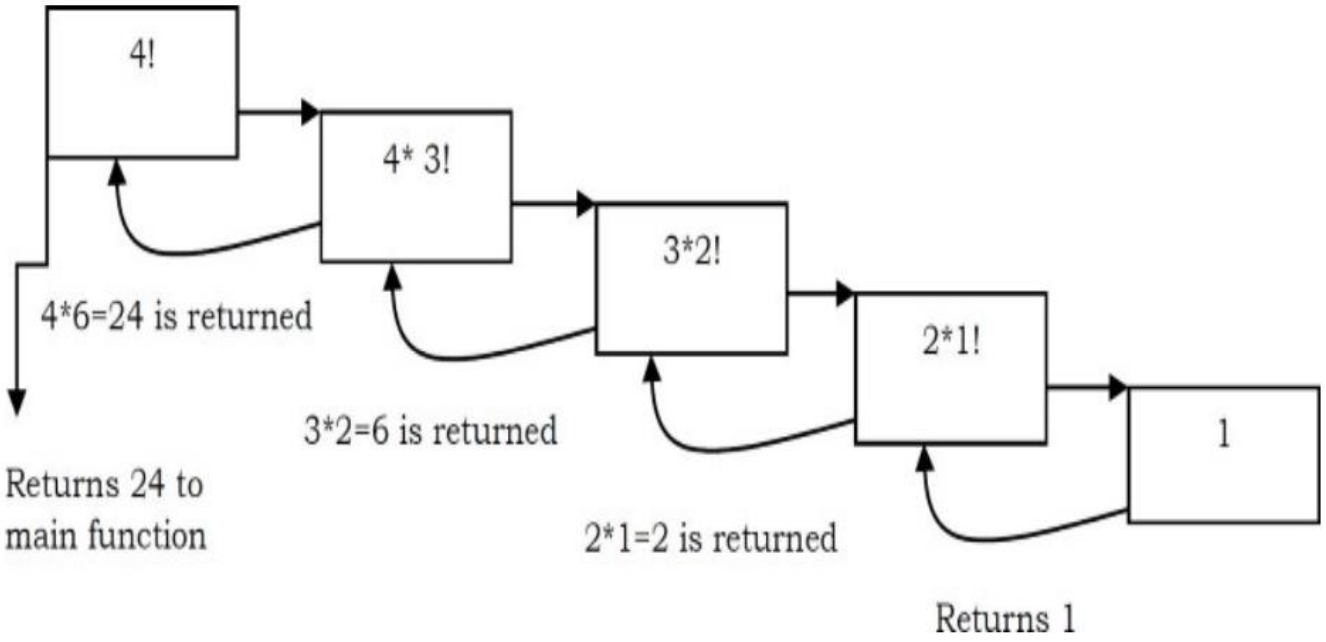
Recursion

- Any function which calls itself is called *recursive*.
- A recursive method solves a problem by calling a copy of itself to work on a smaller problem.
- Recursive code is generally shorter and easier to write than iterative code.
- Generally, loops are turned into recursive functions when they are compiled or interpreted.
- A recursive function performs a task in part by calling itself to perform the subtasks. At some point, the function encounters a subtask that it can perform without calling itself. This case, where the function does not recursive, is called the base case. The former, where the function calls itself to perform a subtask, is referred to as the recursive case.

Recursion

```
def factorial(n):
    if n == 0:      # base case: fact of 0 is 1
        return 1
    else:           # recursive case: multiply n by (n-1) factorial
        return n * factorial(n - 1)

print(factorial(5))
```



- A recursive function uses memory by creating a new stack frame in the memory for each recursive call. The stack frame stores the local variables and parameters for the function call.
- When the function returns, the stack frame is popped off the stack and the memory is freed.

Recursion

Notes on Recursion

- Recursive algorithms have two types of cases, recursive cases and base cases.
- Every recursive function case must terminate at a base case.
- Generally, iterative solutions are more efficient than recursive solutions [due to the overhead of function calls].
- A recursive algorithm can be implemented without recursive function calls using a stack, but it's usually more trouble than its worth. That means any problem that can be solved recursively can also be solved iteratively.
- Some problems are best suited for recursive solutions while others are not.

Recursion

Example Algorithms of Recursion

- Fibonacci Series, Factorial Finding
- Merge Sort, Quick Sort
- Binary Search
- Tree Traversals and many Tree Problems: InOrder, PreOrder PostOrder
- Graph Traversals: DFS [Depth First Search] and BFS [Breadth First Search]
- Dynamic Programming Examples
- Divide and Conquer Algorithms
- Towers of Hanoi

Recursion versus Iteration

- **Recursion**
  - Terminates when a base case is reached.
  - Each recursive call requires extra space on the stack frame (memory).
  - If we get infinite recursion, the program may run out of memory and result in stack overflow.
  - Solutions to some problems are easier to formulate recursively.
- **Iteration**
  - Terminates when a condition is proven to be false.
  - Each iteration does not require extra space.
  - An infinite loop could loop forever since there is no extra memory being created.
  - Iterative solutions to a problem may not always be as obvious as a recursive solution.

Example

- Write a code that prints numbers 1 to n backward.

Example

- Given an array, check whether the array is in sorted order with recursion.

Backtracking

- *Backtracking* is a general algorithmic technique used to find all (or some) solutions to a problem by incrementally building a solution and then undoing it when it's determined that the current path cannot lead to a valid solution.
- Backtracking is an improvement of the *brute force* approach. It systematically searches for a solution to a problem among all available options.
- Backtracking is a form of *recursion*.
- It is commonly used in combinatorial optimization problems, such as the N-Queens problem, Sudoku, and the traveling salesman problem, where you need to explore different possibilities to find a valid solution.

Backtracking

- The basic idea behind backtracking is to:
  1. Start with an empty solution or an initial partial solution.
  2. Extend the solution incrementally, one piece at a time.
  3. If the current solution cannot be extended to a valid solution, backtrack to the previous state and try a different option.
  4. Continue this process until a valid solution is found or all possibilities have been exhausted.

Backtracking versus Recursion

1. **Backtracking**:
    1. **Problem Solving Approach**: Backtracking is a specific problem-solving technique that is used to find all (or some) solutions to a problem by incrementally building a solution and undoing it when a valid solution cannot be found along the current path. It explores multiple possibilities and is particularly useful for combinatorial optimization problems.
    2. **Control Flow**: Backtracking uses a recursive approach to explore different possibilities and backtrack when necessary to explore other paths. It keeps track of the current state and returns to a previous state when a solution is not possible.
    3. **Example**: Solving the N-Queens problem, Sudoku, or finding subsets with a given sum, as shown in the previous examples.
2. **Recursion**:
    1. **Problem Solving Approach**: Recursion is a general programming technique where a function calls itself to solve a problem by breaking it down into smaller, similar subproblems. Each recursive call reduces the problem until a base case is reached, at which point the recursion unwinds.
    2. **Control Flow**: Recursion involves calling the same function with different input parameters. It creates a call stack with multiple function calls that need to be resolved before the recursion can terminate.
    3. **Example**: Calculating the factorial of a number, traversing a tree data structure, or solving the Tower of Hanoi problem.

Example

- Given a set of positive integers and a target sum, determine if there exists a subset of the integers that adds up to the target sum.
- For example, consider the set of numbers {3, 34, 4, 12, 5, 2} and a target sum of 9. You can find a subset {4, 5} that adds up to the target sum.

```
def is_subset_sum_possible(numbers, target, index, current_sum):
  if current_sum == target:
    return True
  if current_sum > target or index == len(numbers):
    return False

  # Include the current number in the subset
  if is_subset_sum_possible(numbers, target, index + 1, current_sum + numbers[index]):
    return True

  # Exclude the current number from the subset
  if is_subset_sum_possible(numbers, target, index + 1, current_sum):
    return True

  return False

def subset_sum(numbers, target):
  return is_subset_sum_possible(numbers, target, 0, 0)

# Example usage:
numbers = [3, 34, 4, 12, 5, 2]
target = 9

if subset_sum(numbers, target):
  print("Subset with sum", target, "exists.")
else:
  print("No subset with sum", target, "exists.")
```

Homework

- We'll assume you have a maze represented as a 2D grid, where you need to find a path from the start to the exit while avoiding obstacles.
  - maze = [[1, 0, 0, 0, 1],
    [1, 1, 0, 1, 1],
    [0, 1, 1, 1, 0],
    [0, 0, 0, 1, 1],
    [0, 0, 0, 0, 1]]
  - 1: Typically, a cell with a value of 1 represents a walkable path or an open cell in the maze. You can move from one cell with a value of 1 to another cell with a value of 1.
  - 0: A cell with a value of 0 usually represents a blocked or obstacle cell. You cannot move through or enter cells with a value of 0.
- The backtracking algorithm's objective is to navigate from the start to the exit while respecting these rules.


Thank you for
your attention